UNIVERSIDAD NACIONAL
SAN AGUSTIN

FACULTAD DE PRODUCCION Y SERVICIOS
ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

Informe sobre KD Tree Search

Estructuras de Datos Avanzadas

EDUARDO ANTONIO SANCHEZ HINCHO
LUIS GUILLERMO VILLANUEVA FLORES

AREQUIPA
2019

1. Introducción

En ciencias de la computación, un Árbol kd (abreviatura de árbol k-dimensional) es una estructura de datos de particionado del espacio que organiza los puntos en un Espacio euclídeo de k dimensiones. Los árboles kd son un caso especial de los árboles BSP. Un árbol kd emplea sólo planos perpendiculares a uno de los ejes del sistema de coordenadas. Esto difiere de los árboles BSP, donde los planos pueden ser arbitrarios. Además, todos los nodos de un árbol kd, desde el nodo raíz hasta los nodos hoja, almacenan un punto. .

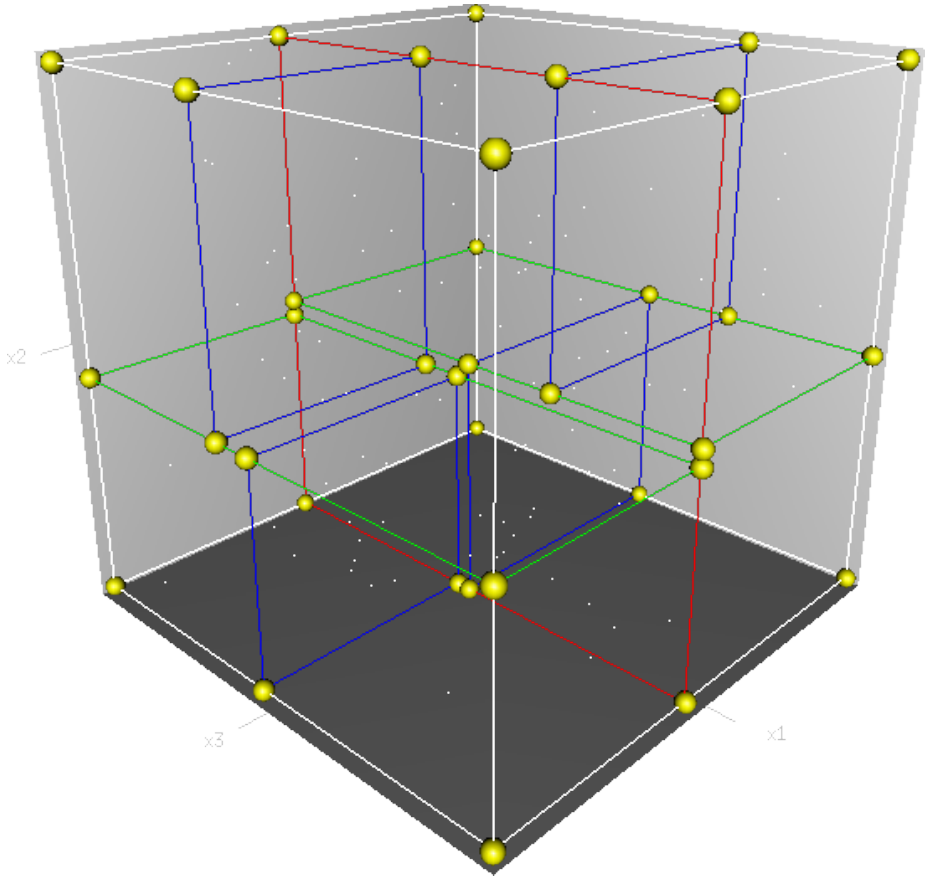

Figura 1: Representación del KD Tree en 3D

1.1. Usos del KD Tree

- Búsqueda ortogonal en un árbol kd: Usar un árbol kd para encontrar todos los puntos que se encuentran en un rectángulo determinado (o análogo de más dimensiones). Esta operación también se denomina rango de búsqueda ortogonal.
- Determinar dónde evaluar una superficie: En las regresiones locales es común evaluar la superficie contenida directamente sólo por los vértices del árbol kd e interpolar en algún punto. Este uso, reflejado en la imagen de arriba, busca asegurar que sólo se realizarán las evaluaciones directas necesarias. Como los árboles kd se .ªdaptan.ªl espacio, este método puede suministrar una excelente aproximación a las verdaderas superficies de regresión local.

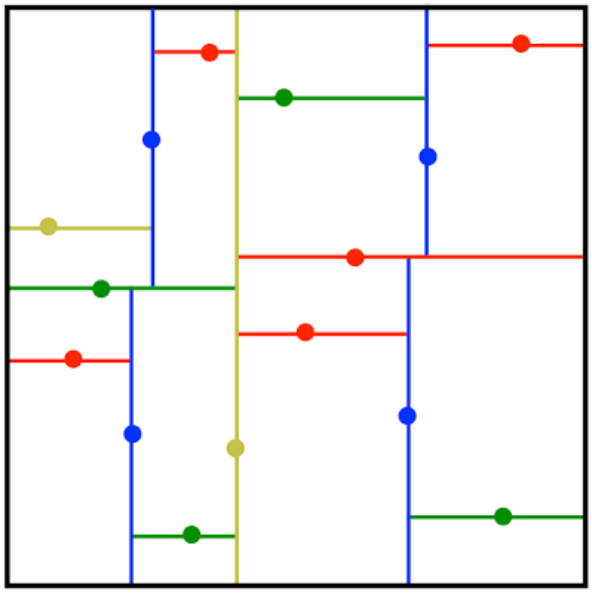

Figura 2: Representación del KD Tree en 2D

2. KD Tree Search

Para este punto necesitamos conocer la distacia ,para ello usamos la distancia euclidiana , que se muestra en código a continuación.

```
function distanceSquared(point1, point2){
  //console.log(point1,point2);
  var distance = 0;
  for (var i = 0; i < k; i++)
    distance += Math.pow((point1[i] - point2[i]), 2);
  return Math.sqrt(distance);
}
```

Listing 1: Distancia Euclidiana

Para la búsqueda de un punto más cercano a los puntos que coniene nuestro KD tree, se pueden implementar diferentes algoritmos como ya vimos, uno de ellos es el de fuerza bruta que se muestra a continuación.

```
function closest_point_brute_force(points, point){

  var m=distanceSquared(points[0],point);
  var pointm=points[0];
  for (var i = 1; i < points.length; i++) {
    var aux=distanceSquared(points[i],point);
    if(aux<m){
      m=aux;
      pointm=points[i];
    }
  }
  return pointm;
}
```

Listing 2: Búsqueda de un punto cercano por fuerza bruta

También vimos otra función que era más eficiente a la anterior pero con la restricción que no cunmplía para todos los casos, la mosttramos a continuación.

```
function naive_closest_point2(node, point, depth = 0, best = null){

  if(node==null)return best;

  var axis=depth%k;
  var next_best=null;
  var next_branch=null;
  if(best==null||distanceSquared(best, point)> distanceSquared(node
      .point,point))
    next_best=node.point;
  else
    next_best=best;
  if(point[axis]<node.point[axis])
    next_branch=node.left;
  else
    next_branch=node.right;
  return naive_closest_point2(next_branch,point,depth+1,next_best);
}
```

Listing 3: Búsqueda de un punto cercano por algoritmo mejorado

Después de esta función implementamos la función closer_point que lo que hace es comparar 3 puntos y retorna el mejor , osea el más cercano en los tres, hicimos una variación agregándole a nuestra clase node un aributo llamado visi, que nos dice si un nodo ya ha sido visitado, eso para cuando querramos consultar el k nodos más cercanos. Aquí el fragmento de código:

```
function closer_point(point,p1,p2){
  if(p1==null && p2==null)return;
  if(p1==null)
    return p2;
  if(p2==null)
    return p1;

  if(distanceSquared(point,p1.point)<=distanceSquared(point,p2.
      point))
    return p1;
  return p2;
}
```

Listing 4: Función closer_point

Ahora para poder mejorar nuestro algoritmo lo que hacemos es creamos nuesra clase cola de prioridad, para que por medio de esta estructura podamos optimizar el código.

```
class Cola{
  constructor(cant){
    this.n=cant;
    this.lis=[];
  }
  agregar(distancia,x){
    for(var i=0;i<this.lis.length;i++){
      if(this.lis[i][1]==x)return;
    }
    if(this.lis.length==this.n){
      if(this.lis[this.n-1][0]>distancia){
        this.lis[this.n-1]=[distancia,x];
        this.lis.sort(function(a,b){
          return a[0]-b[0];
        });
      }
    }
    else{
      this.lis.push([distancia,x]);
      this.lis.sort(function(a,b){
        return a[0]-b[0];
      });
    }

  }
  mostrar(){
    console.log(this.lis)
  }
  top(){
    return this.lis[this.lis.length-1][0];
  }
  llena(){
    if(this.lis.length!=this.n)return false;
    return true;
  }
}
```

Listing 5: Clase Cola de prioridad

Ahora lo que hacemos es modificar un poco nuestra funcion closest_points para que funcione con nuesra cola de prioridad.

```
function closest_point(node, point, depth = 0,cola){
  if (node == null)
    return;

  var axis = depth % k;
  var next_branch = null; //next node brach to look for
  var opposite_branch = null; //opposite
  cola.agregar(distanceSquared(point,node.point),node.point);
  if (point[axis] < node.point[axis]){
    next_branch = node.left;
    opposite_branch = node.right;
  }else{
    next_branch = node.right;
    opposite_branch = node.left;
  }


  //var best=closer_point(point,closest_point(next_branch,point,
      depth+1,cola),node);
  //cola.agregar(distanceSquared(point,best.point),best.point);
  closest_point(next_branch,point,depth+1,cola);
  if(!cola.llena() || cola.top()>Math.abs(point[axis]-node.point[
      axis]))
  {
    //best=closer_point(point,closest_point(opposite_branch,point,
        depth+1,cola),node);
    //best=closer_point(point,closer_point(point,closest_point(
        opposite_branch,point,depth+1,cola),node),best);
    //cola.agregar(distanceSquared(point,best.point),best.point);
    closest_point(opposite_branch,point,depth+1,cola)
  }
}
```

Listing 6: Función closest_points mejorada

Ahora mostramos nuestra función nearest que lo que hace es simplemente encontrar los k puntos que solicitamos.

```
function nearest(root,point,count){
  var cola= new Cola(count);
  closest_point(root,point,0,cola)
  //cola.agregar(0,[5,2])
  return cola.lis;

}
```

Listing 7: Función closest_points mejorada

3. Prueba del código

Probamos nuestro código consultamos 12 puntos que han sido generados aleatoriamente y consultamos los más cercanos al punto $(140, 190)$, para ello hemos hecho este código que nos muestra en pantalla el punto que queremos y nos pinta los más cercanos a ese , que para este caso hemos pedido 4 más cercanos.

```
  var data=[]

  var point = [140,90];
  for(let i = 0; i < 12; i++){
    var x = Math.floor(Math.random() * height);
    var y = Math.floor(Math.random() * height);
    data.push([x, y]);
    fill(255, 255, 255);
    circle(x, height - y, 7);
    textSize(8);
    text(x + ',' + y, x + 5, height - y);
  }
  var root = build_kdtree(data,0);
  console.log(root);
  var l=generate_dot(root);
  var cerca=nearest(root,point,4);
  fill(0,255,255);
  x=point[0];
  y=point[1];
  circle(x,height-y,7);
  for(let i = 0; i < cerca.length; i++){
    fill(255,0,255);
    x=cerca[i][1][0];
    y=cerca[i][1][1];
    circle(x,height-y,7);
  }

}
```

Listing 8: Insertar y dibujar puntos en pantalla

Podemos corroborar que si lo hizo correctamente.

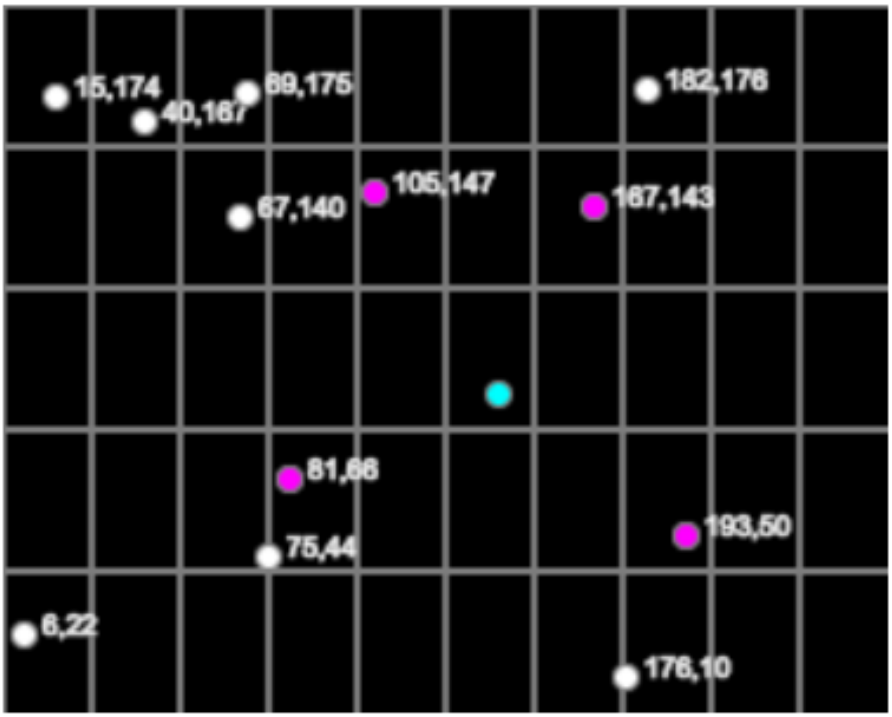

Figura 3: Gráfico donde se muestran lo puntos más cercanos

Ahora si consultamos por consola nos sale lo mismo como se puede observar.

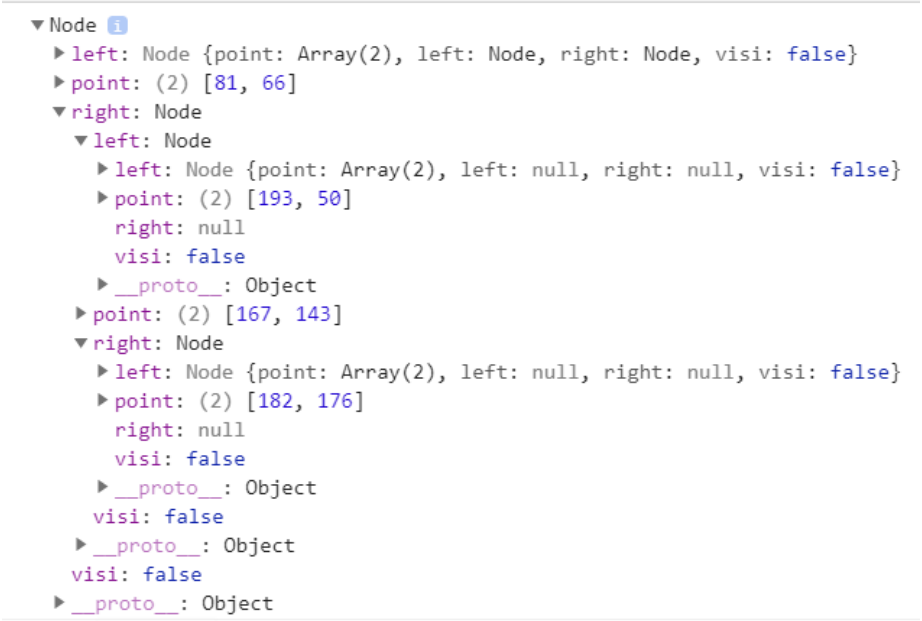

Figura 4: Cuatro puntos más cercanos a $(140, 90)$

Cómo podemos darnos cuenta esos 4 son los más cercanos y en consola nos muestra el valor del nodo porque eso fue lo que cambiamos, y por ende nos muestra el punto.

4. Conclusiones

- Nos dimos cuenta que al igual que en el Quadtree y Octree esta estructura nos puede ayudar mucho al momento de querer distribuir datos de una manera no uniforme, ya que pueden estar dispersos en cualquier espacio donde nosotros lo definamos y querramos que esten.
- Vimos que hay muchas maneras de hacer una búsqueda en un kd tree, pero podemos encontrarnos con algoritmos ineficientes, y eso por consecuencia nos traera problemas cuando el temaño de puntos sea mayor.
- Nos dimos cuenta que el kd tree también es bien útil en la busqueda de un punto en nuestro kd tree y eso puede tener diversas aplicaciones para proyectos futuros.
- Vimos que hay una similitud con respecto al Quadtree y Octree porque en ambos casos uno puede ser en 2D y en 3D respectivamente pero en este caso puede adaptarse para esas dimensiones y para más si en caso hubiera.
- Nos dimos cuenta que en el KDTree es mucho más fácil el almacenamiento de los datos respecto a las anteriores estructuras trabajadas.

5. Referencias

- Árbol kd. (2019). Retrieved 22 November 2019, from https://es.wikipedia.org/wiki/
- K Dimensional Tree — Set 1 (Search and Insert) - GeeksforGeeks. (2019). Retrieved 22 November 2019, from https://www.geeksforgeeks.org/k-dimensional-tree/